dongguk UNIVERSITY

CSE 2017 Data Structures and Lab

Lecture #4: Linked Stack

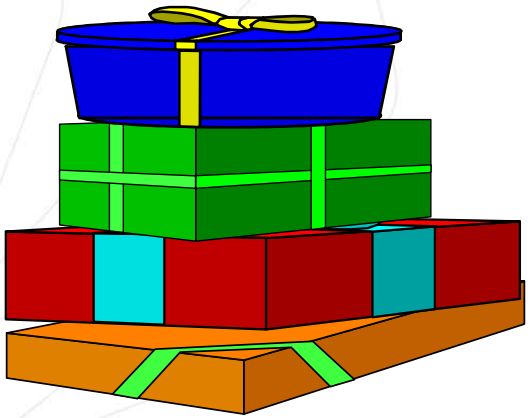
Linked List

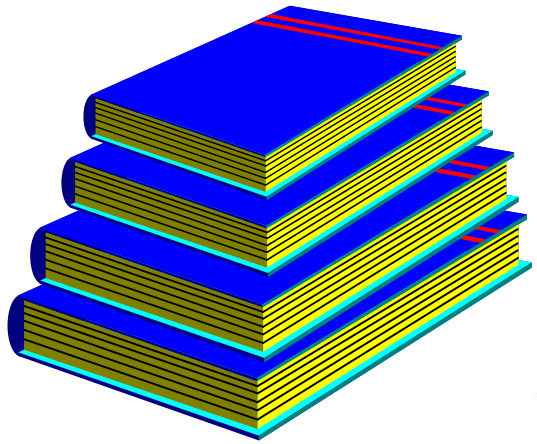
Eun Man Choi

What is a stack?

- It is an ordered group of homogeneous items.
- Items are added to and removed from the top of the stack

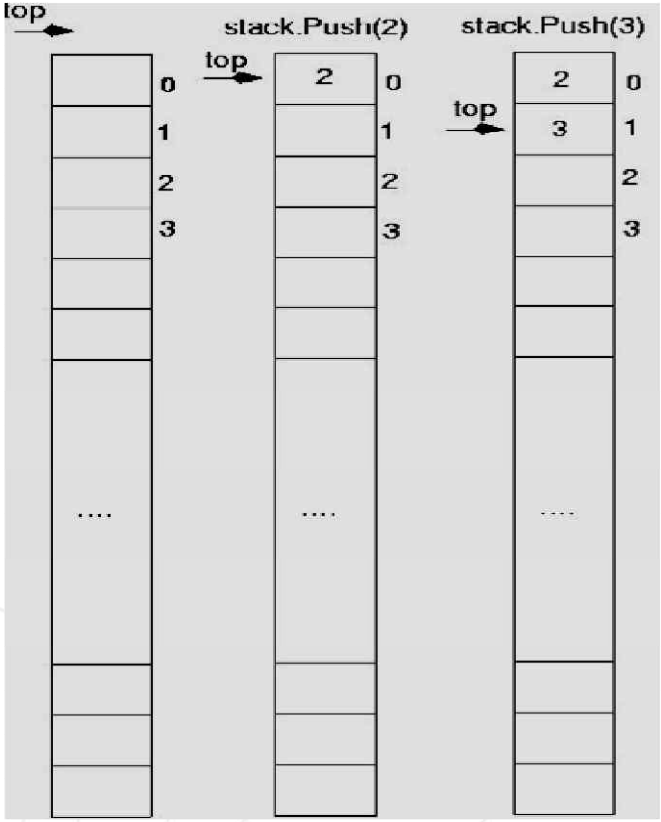
  LIFO property: Last In, First Out
- The last item added would be the first to be removed



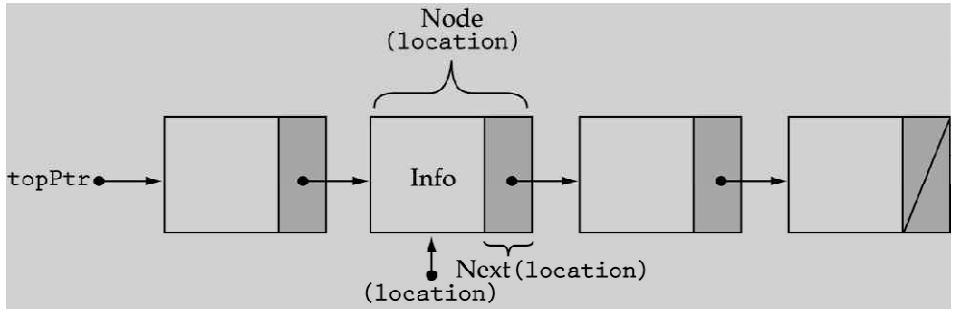


Stack Implementations

Array-based

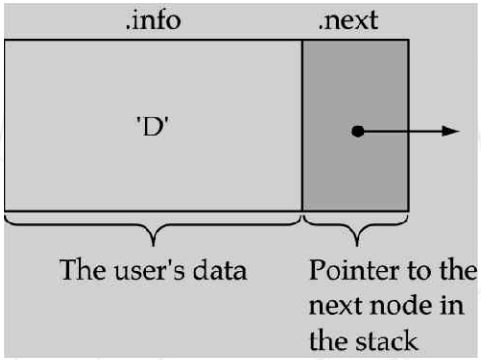


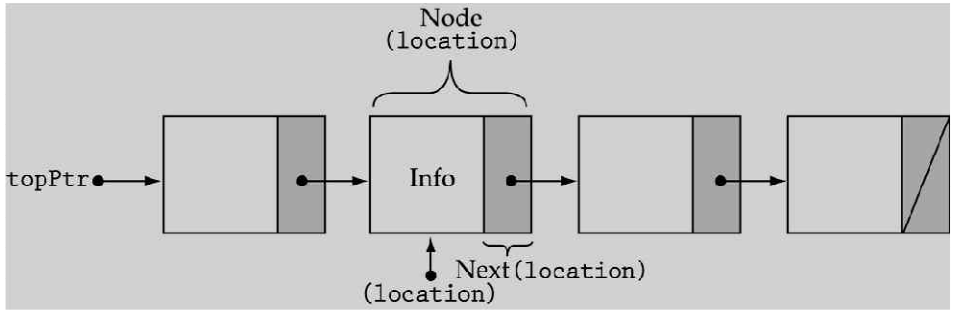
Linked-list-based



Linked-list-based Stacks

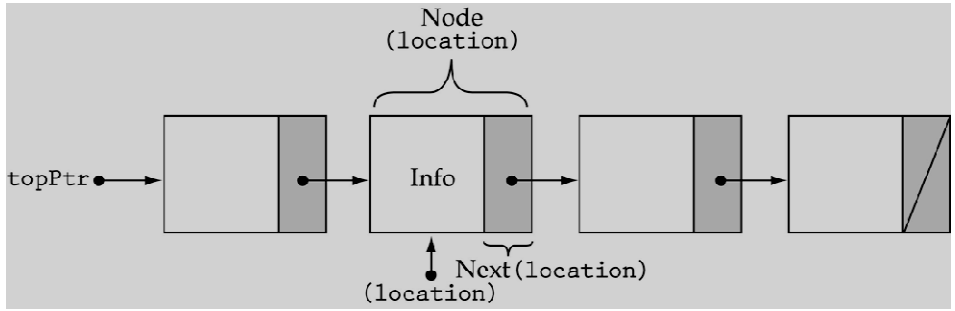
```
template<class ItemType>
struct NodeType<ItemType> {
    ItemType info;
    NodeType<ItemType>* next;
};
```





Linked-list-based Stacks (cont'd)

```
template<class ItemType>
struct NodeType<ItemType>;


template<class ItemType>
class StackType {
 public:
    StackType();
    ~StackType();
    void MakeEmpty();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType);
    void Pop(ItemType&);
 private:
    NodeType<ItemType>* topPtr;
};
```



Linked-list-based Stacks (cont'd)

```
template<class ItemType>
StackType<ItemType>::StackType() {
 topPtr = NULL;
}
```

O(1)

```
template<class ItemType>
void StackType<ItemType>::MakeEmpty() {
 NodeType<ItemType>* tempPtr;

 while(topPtr != NULL) {
    tempPtr = topPtr;
    topPtr = topPtr->next;
    delete tempPtr;
 }
}
```

O(N)

Linked-list-based Stacks (cont'd)

```
template<class ItemType>
StackType<ItemType>::~StackType(){
  MakeEmpty();
}
```

O(N)

```
template<class ItemType>
bool StackType<ItemType>::IsEmpty() const {
 return(topPtr == NULL);
}
```

O(1)

Linked-list-based Stacks (cont'd)

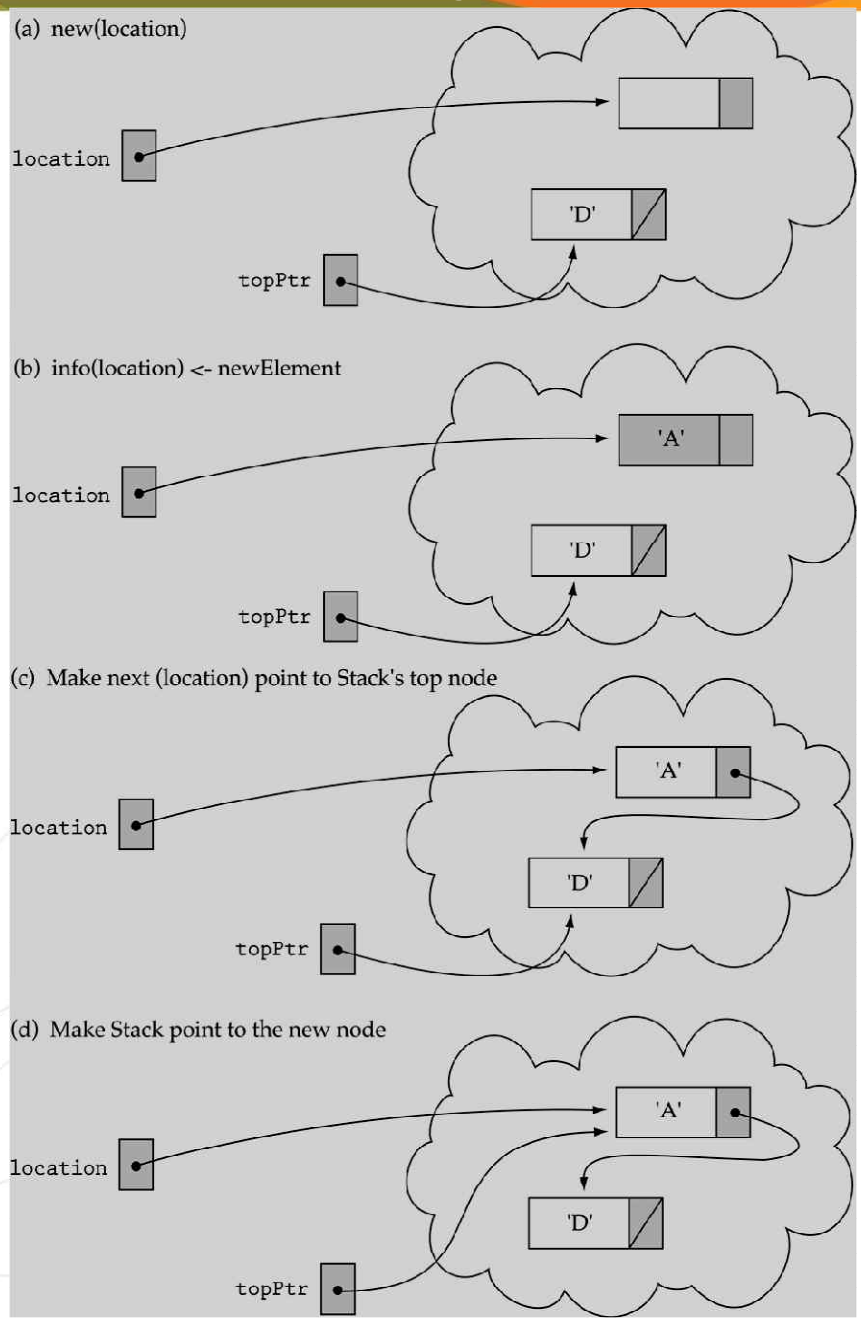
```
template<class ItemType>
bool StackType<ItemType>::IsFull() const {
 NodeType<ItemType>* location;

 location = new NodeType<ItemType>; // test
 if(location == NULL)
   return true;
 else {
   delete location;
   return false;
 }
}
```

O(1)

Push (ItemType newItem)

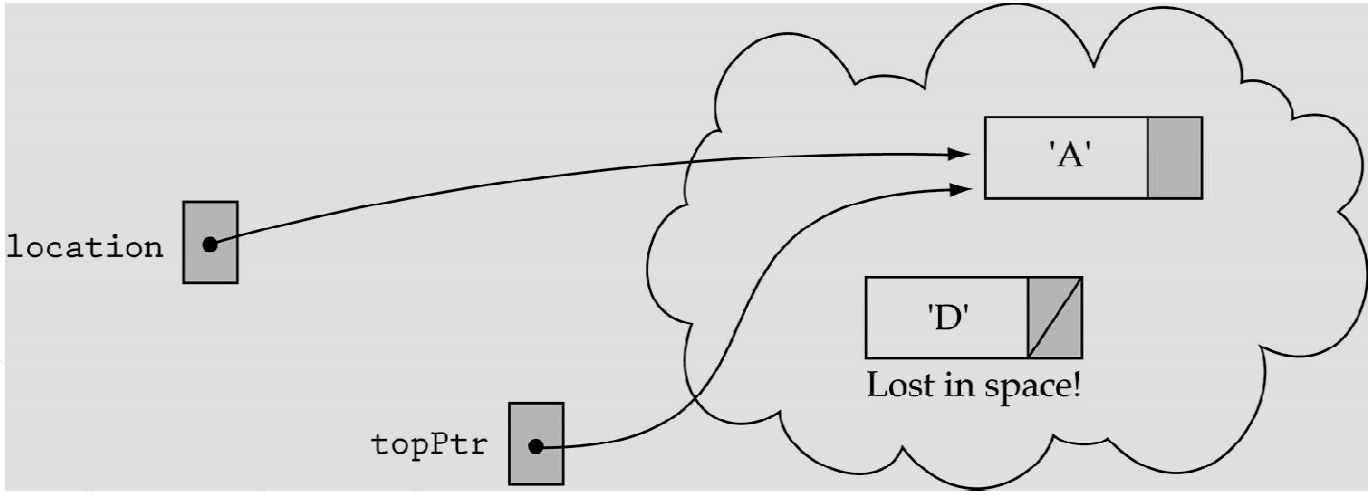
- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

Pushing on a non-empty stack

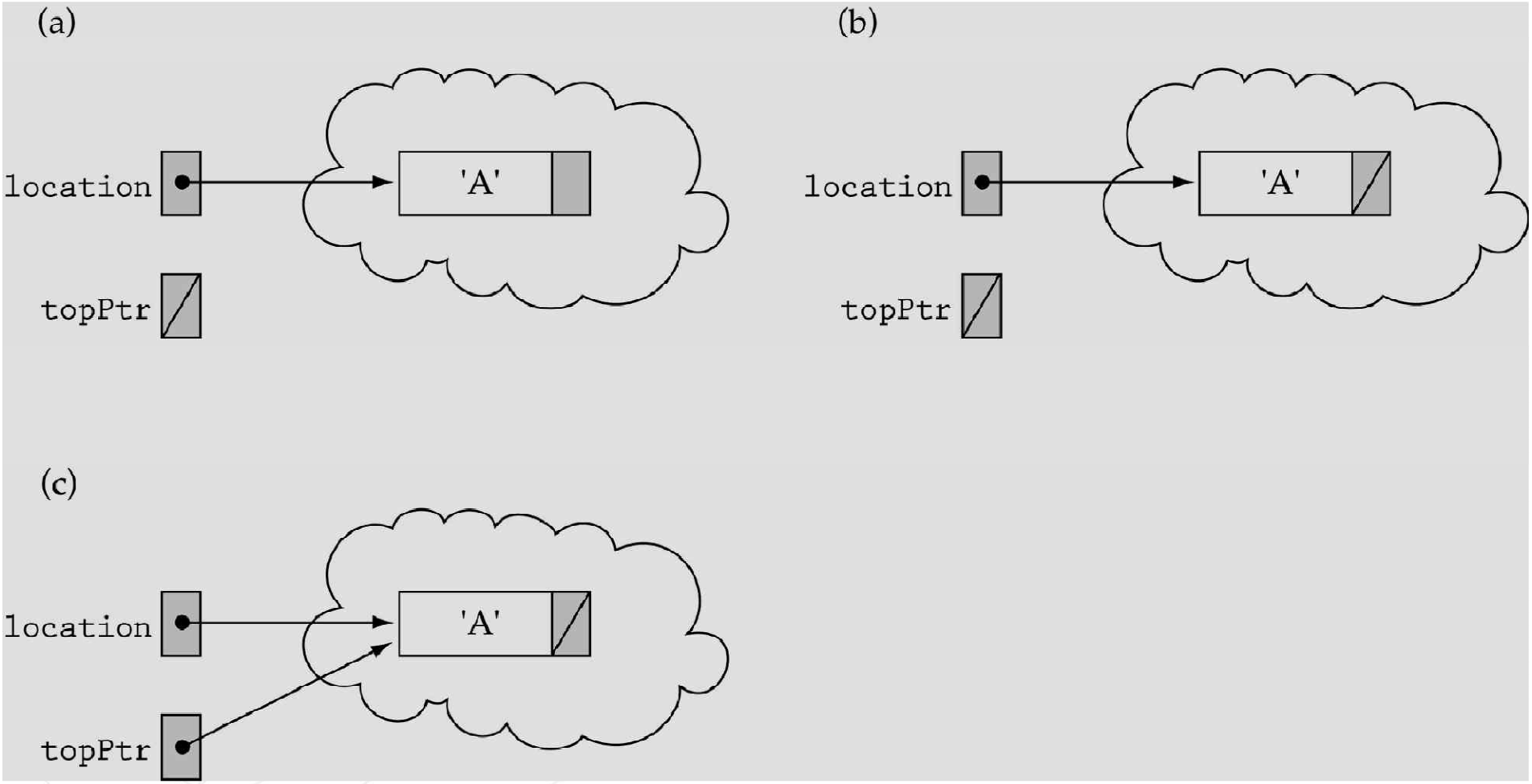


Pushing on a non-empty stack (cont.)

- The order of changing the pointers is important!



Special Case: pushing on an empty stack



Function Push

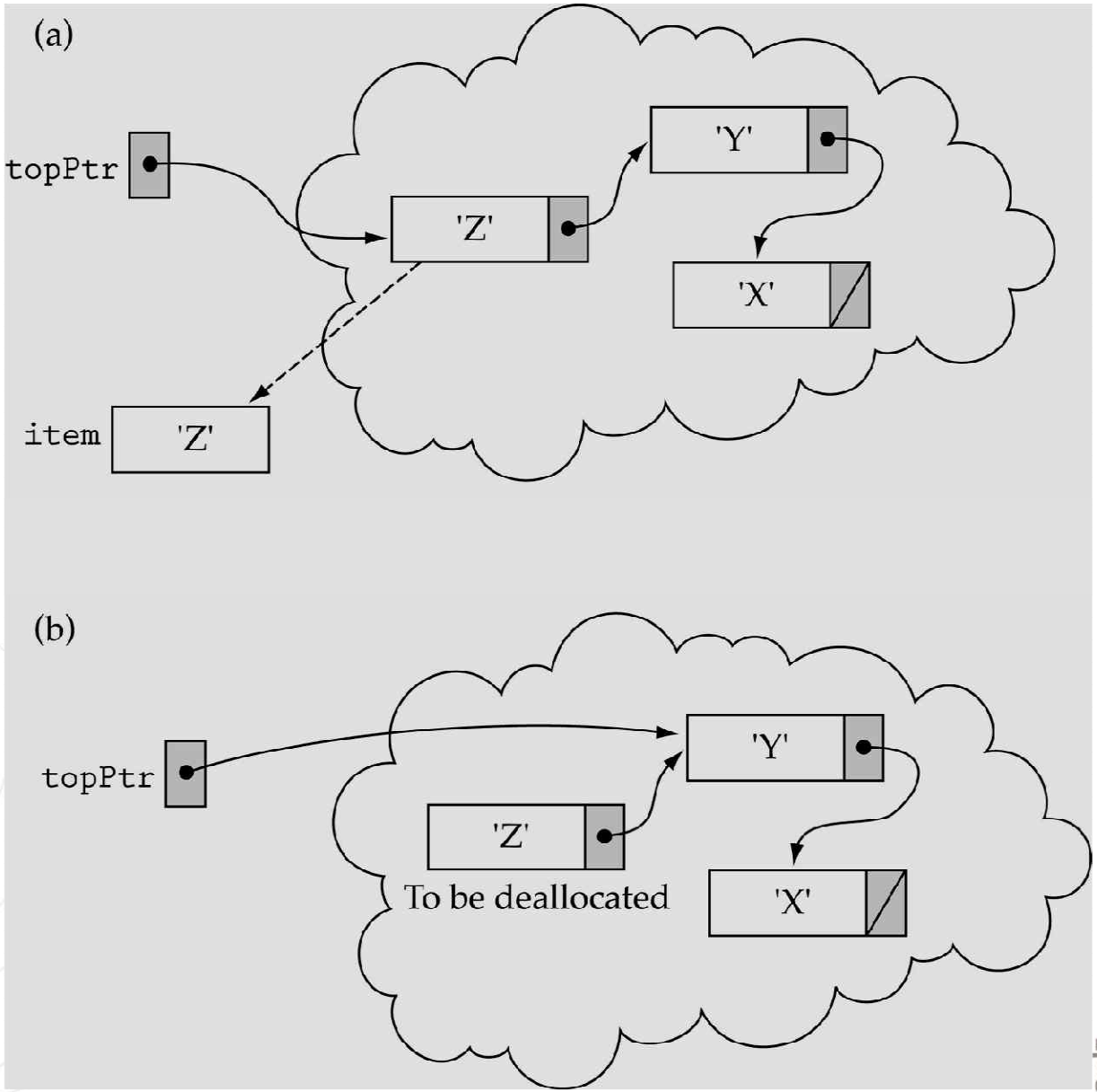
```
template <class ItemType>
void StackType<ItemType>::Push(ItemType item)
{
 NodeType<ItemType>* location;

 location = new NodeType<ItemType>;
 location->info = newItem;
 location->next = topPtr;
 topPtr = location;
}
```

O(1)

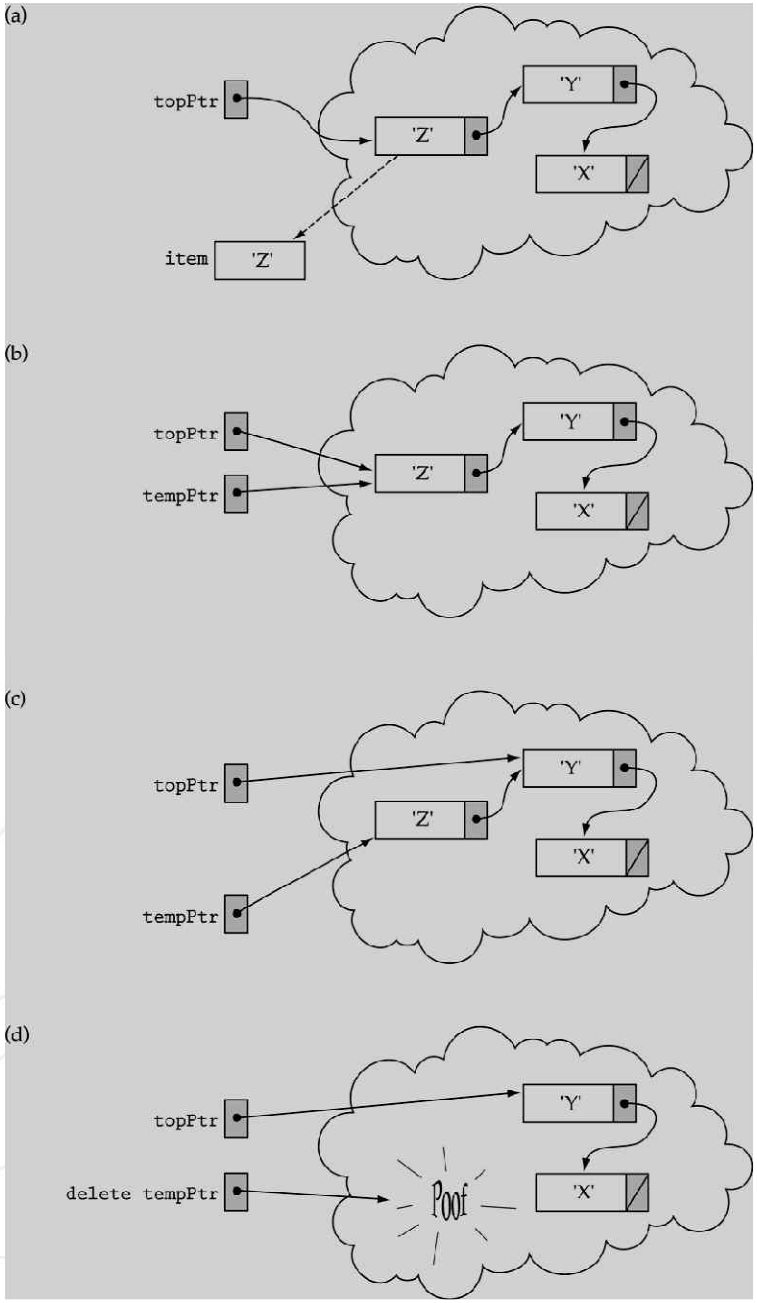
Pop (ItemType& item)

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.

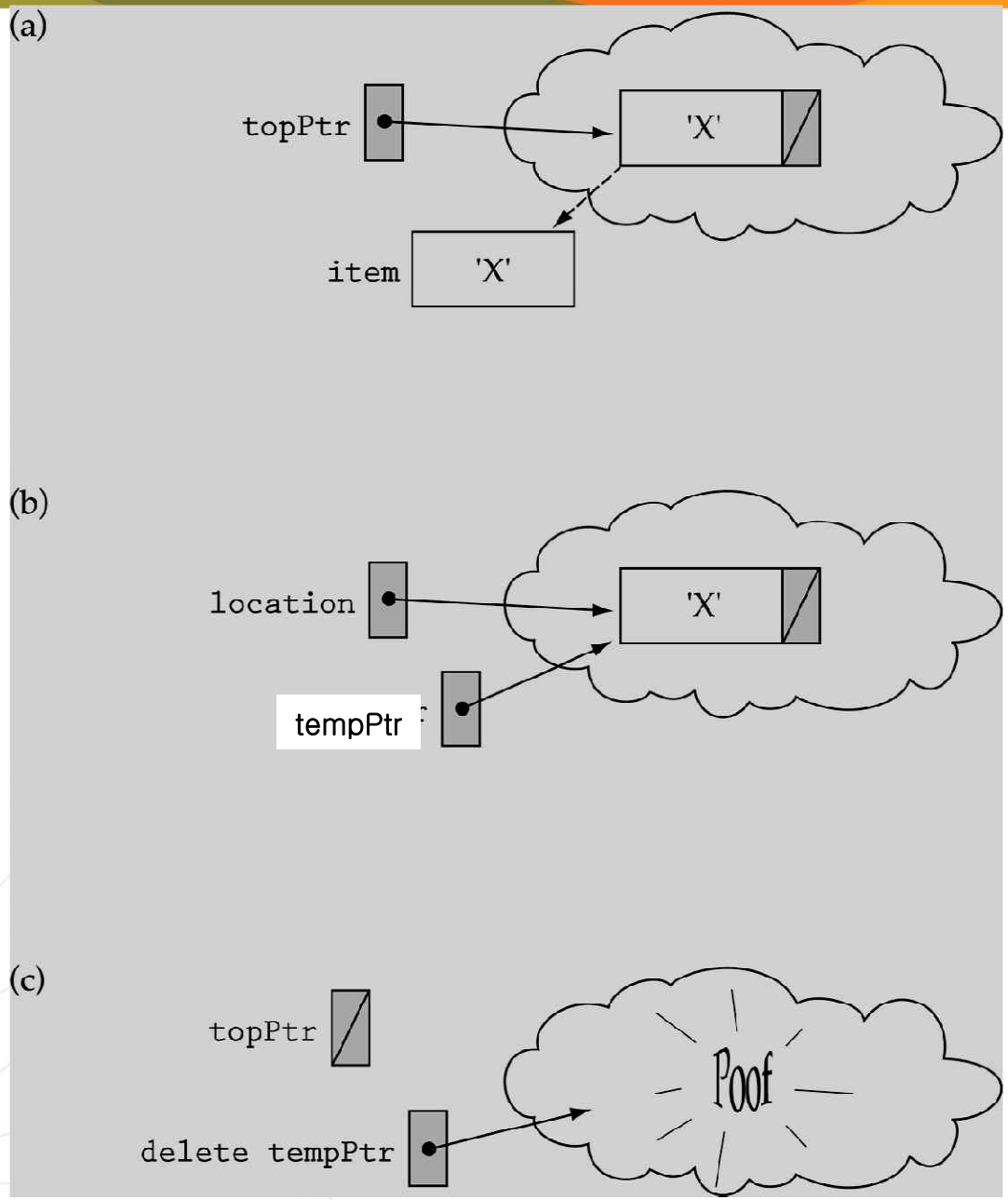
Popping the top element



Popping the top element(cont.)



Special case: popping the last element on the stack



Function Pop

```
template <class ItemType>
void StackType<ItemType>::Pop(ItemType& item)
{
 NodeType<ItemType>* tempPtr;

 item = topPtr->info;
 tempPtr = topPtr;
 topPtr = topPtr->next;
 delete tempPtr;
}
```

O(1)

Comparing stack implementations

Big-O Comparison of Stack Operations		
Operation	Array Implementation	Linked Implementation
Constructor	O(1)	O(1)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
IsEmpty	O(1)	O(1)
Push	O(1)	O(1)
Pop	O(1)	O(1)
Destructor	O(1)	O(N)

Array-vs Linked-list-based Stack Implementations

- Array-based implementation is simple but:
  - The size of the stack must be determined when a stack object is declared.
  - Space is wasted if we use less elements.
  - We cannot “push" more elements than the array can hold.
- Linked-list-based implementation alleviates these problems but time requirements might increase.

Using stacks: evaluate postfix expressions

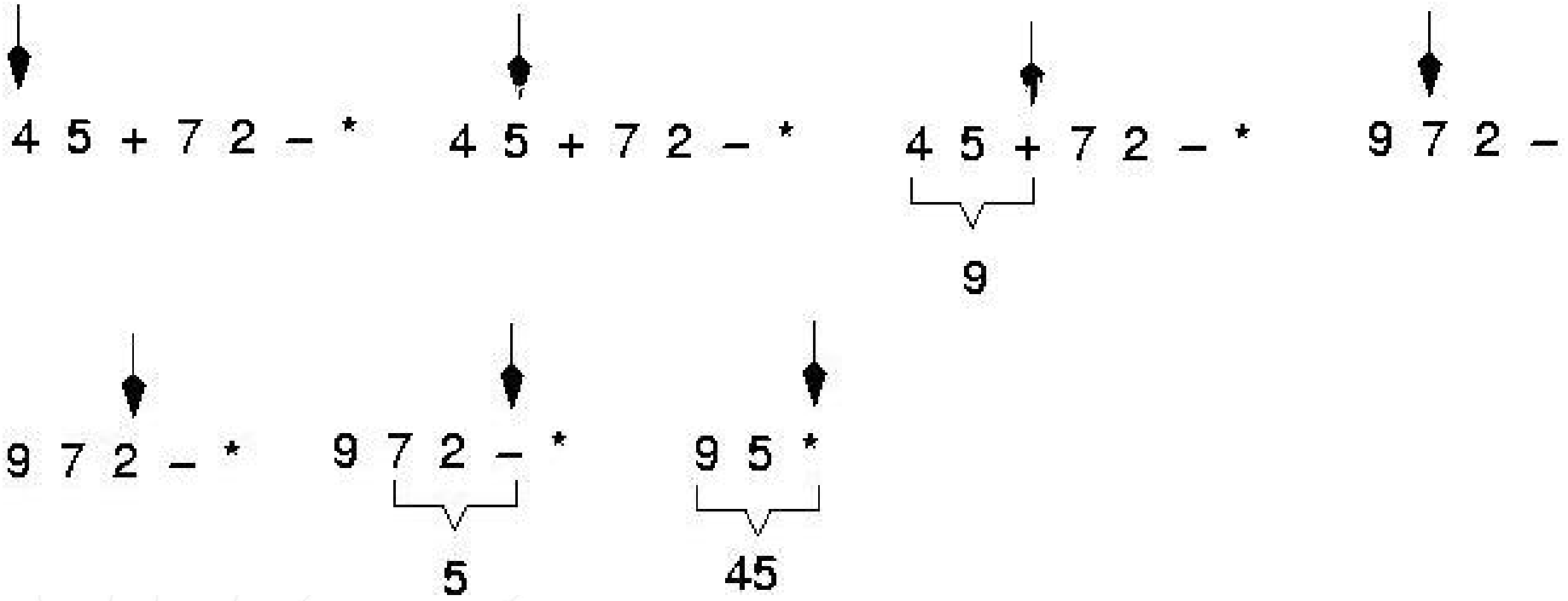
- Postfix notation is another way of writing arithmetic expressions.
- In postfix notation, the operator is written after the two operands.

infix: 2+5 *postfix*: 2 5 +

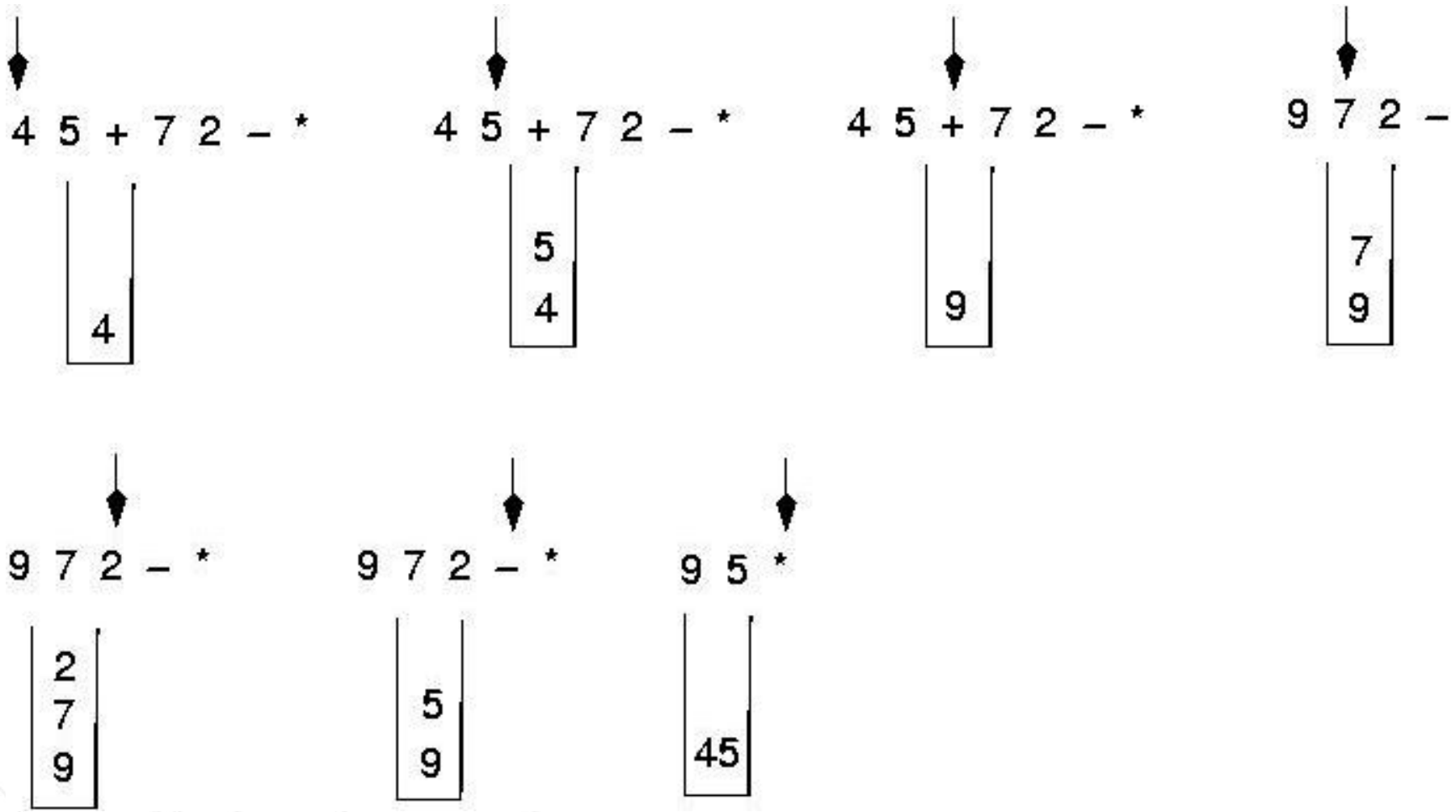
- Why using postfix notation?

Precedence rules and parentheses are not required!

Example: postfix expressions(cont.)



Postfix expressions: Algorithm using stacks



Exercise

- **Write the body for a client function that replaces each copy of an item in a stack with another item. Use the following specification.**

ReplaceItem(StackType& stack, ItemType oldItem, ItemType newItem)

Function: **Replaces all occurrences of oldItem with newItem.**

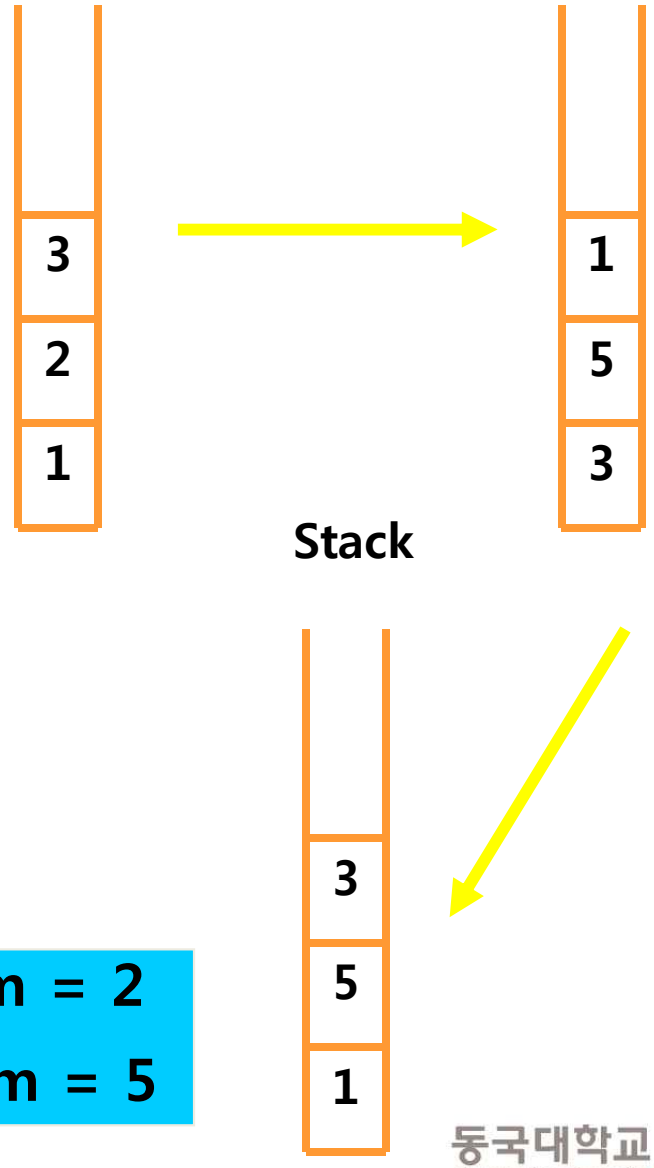
Precondition: **stack has been initialized.**

Postconditions: **Each occurrence of oldItem in stack has been replaced by newItem. Order of other elements remains unchanged.**

Warning: you may not assume any knowledge of how the stack is implemented!

ReplaceItem in Stack

```
{
   ItemType item;
   StackType tempStack;

   while (!Stack.IsEmpty())   {
       Stack.Pop(item);
       if (item==oldItem)
           tempStack.Push(newItem);
       else
           tempStack.Push(item);
   }
   while (!tempStack.IsEmpty()) {
       tempStack.Pop(item);
       Stack.Push(item);
   }
}
```



oldItem = 2

newItem = 5

ReplaceItem in Stack

```
{
   ItemType item;
   StackType tempStack;

   while (!Stack.IsEmpty())   {
      Stack.Pop(item);
      if (item==oldItem)
         tempStack.Push(newItem);
      else
         tempStack.Push(item);
   }
   while (!tempStack.IsEmpty()) {
      tempStack.Pop(item);
      Stack.Push(item);
   }
}
```

What are the time requirements using big-O?

O(N)

ADT Unsorted List Operations with Linked Structure

Transformers

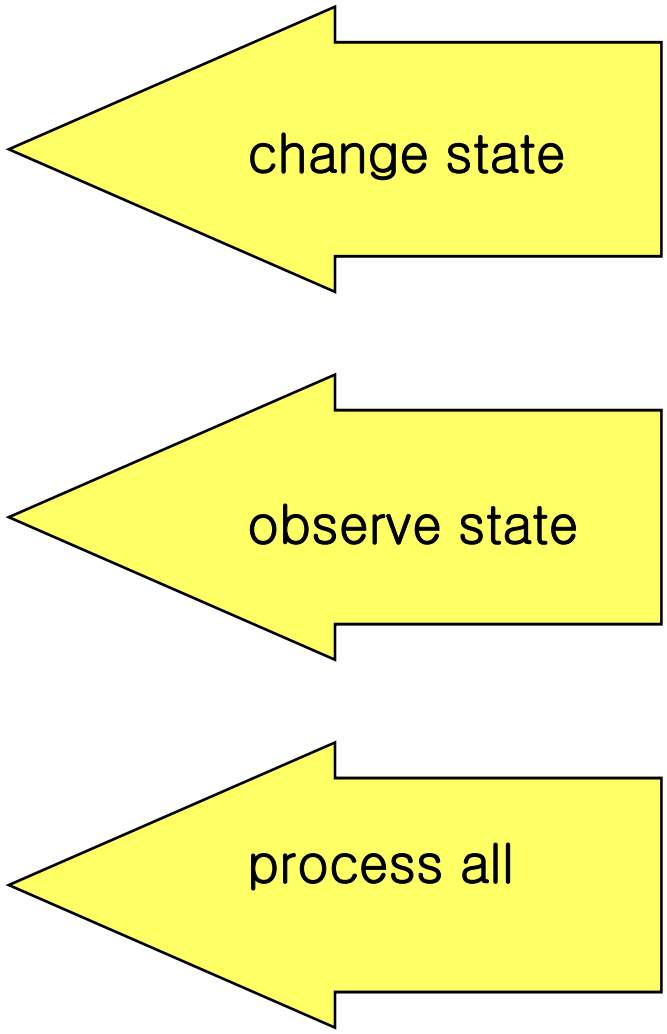
- MakeEmpty
- InsertItem
- DeleteItem

Observers

- IsFull
- LengthIs
- RetrieveItem

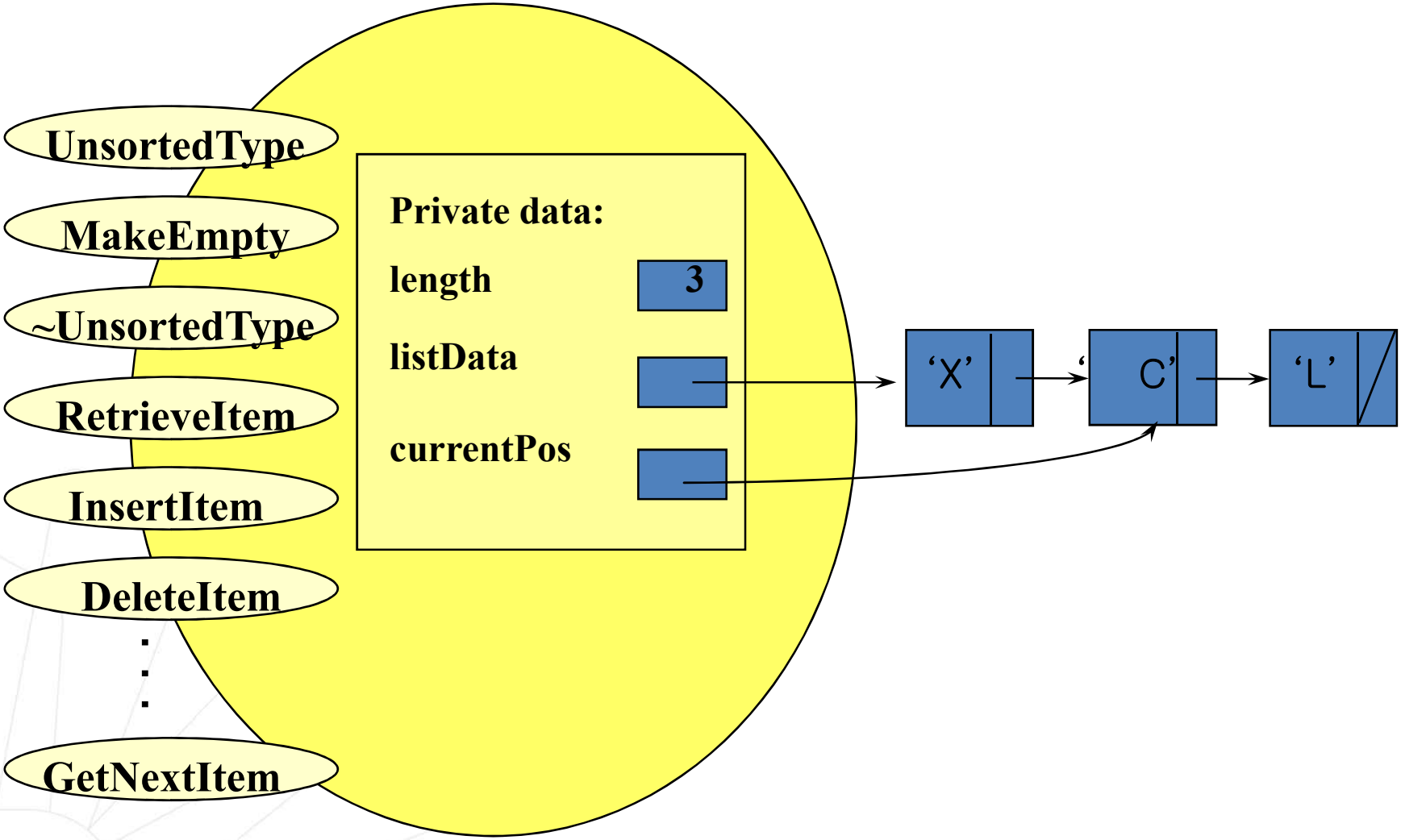
Iterators

- ResetList
- GetNextItem



```
#include  "ItemType.h"                    //  unsorted.h
  . . .
template <class ItemType>
class  UnsortedType
{
public :                                  //  LINKED LIST IMPLEMENTATION
  UnsortedType ( ) ;
  ~UnsortedType ( ) ;
  void          MakeEmpty ( ) ;
  bool           IsFull ( )  const ;
  int            LengthIs ( )  const ;
  void          RetrieveItem ( ItemType&  item, bool&  found ) ;
  void          InsertItem ( ItemType  item ) ;
  void          DeleteItem ( ItemType  item ) ;
  void          ResetList ( );
  void          GetNextItem ( ItemType&  item ) ;
private :
  NodeType<ItemType>*  listData;
  int     length;
  NodeType<ItemType>*  currentPos;
} ;
```

class UnsortedType<char>



```
// LINKED LIST IMPLEMENTATION  ( unsorted.cpp )
#include "itemtype.h"

template <class ItemType>
UnsortedType<ItemType>::UnsortedType ( )       // constructor
// Pre:          None.
// Post:List is empty.
{
  length  =  0 ;
  listData = NULL;
}

template <class ItemType>
int  UnsortedType<ItemType>::LengthIs ( )  const
// Post:  Function value = number of items in the list.
{
  return  length;
}
```

O(1)

30

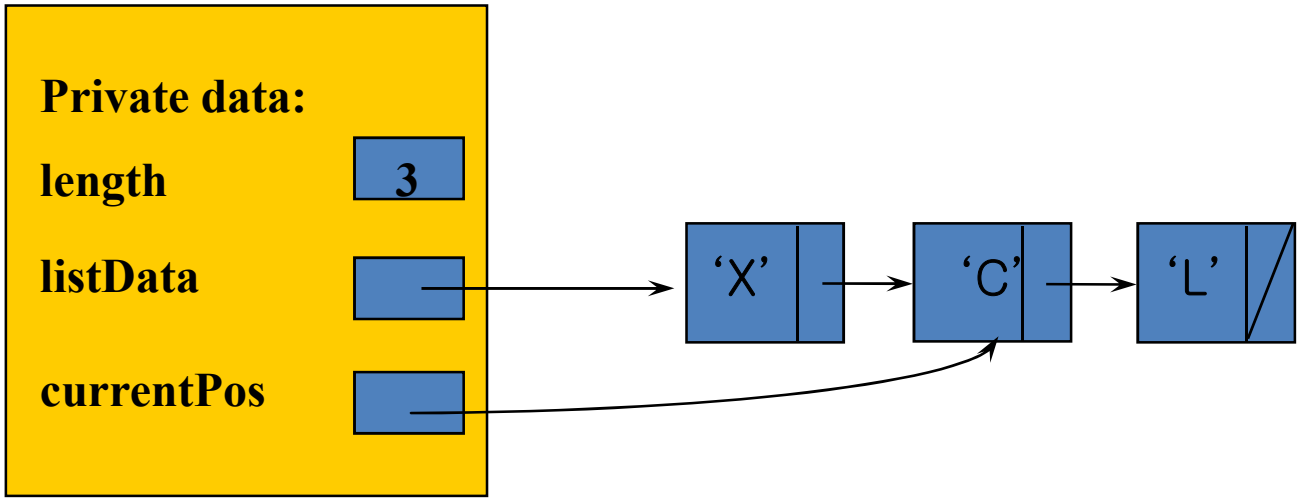
```
template <class ItemType>
void  UnsortedType<ItemType>::RetrieveItem( ItemType&  item, bool&
  found )
// Pre:   Key member of item is initialized.
// Post:  If found, item's key matches an element's key in the list and a copy
//        of that element has been stored in item; otherwise, item is unchanged.
{   bool  moreToSearch ;
  NodeType<ItemType>*  location ;
    location = listData ;
  found = false ;
  moreToSearch = ( location  !=  NULL ) ;
  while ( moreToSearch  &&  !found )
  {       if ( item  == location->info )              // match here
                  {    found = true ;
                item  = location->info ;
          }
          else                                        // advance pointer
          {   location = location->next ;
              moreToSearch = ( location  !=  NULL ) ;
                  }
    }
```

O(N)

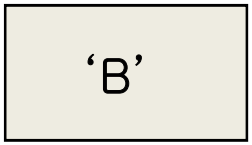
```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem ( ItemType  item )
// Pre:     list is not full and item is not in list.
// Post:    item is in the list;  length has been incremented.
{
  NodeType<ItemType>*  location ;
                                          // obtain and fill a node
    location = new  NodeType<ItemType> ;
  location->info = item ;
  location->next = listData ;
  listData = location ;
  length++ ;
}
```

O(1)

Inserting 'B' into an Unsorted List

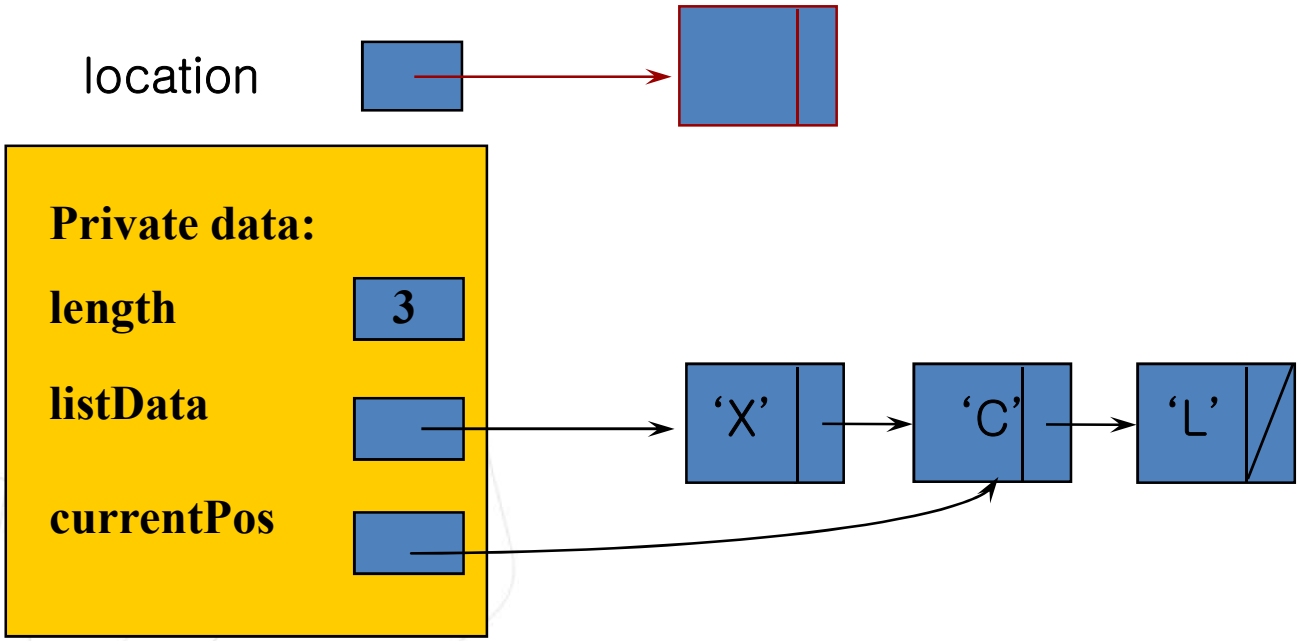


Inserting 'B' into an Unsorted List

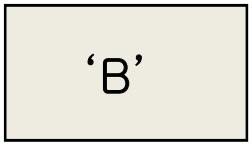


item

```
location = new  NodeType<ItemType>;
```

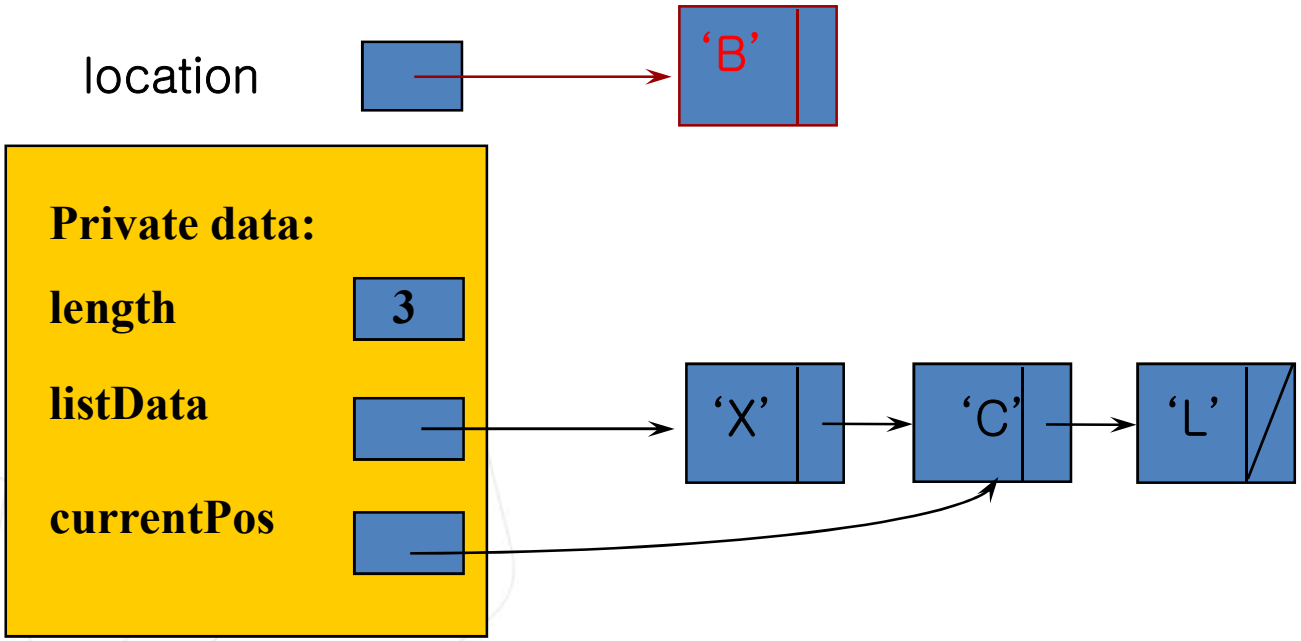


Inserting 'B' into an Unsorted List

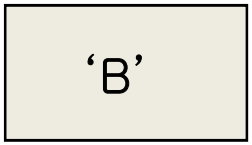


item

```
location->info  =  item ;
```

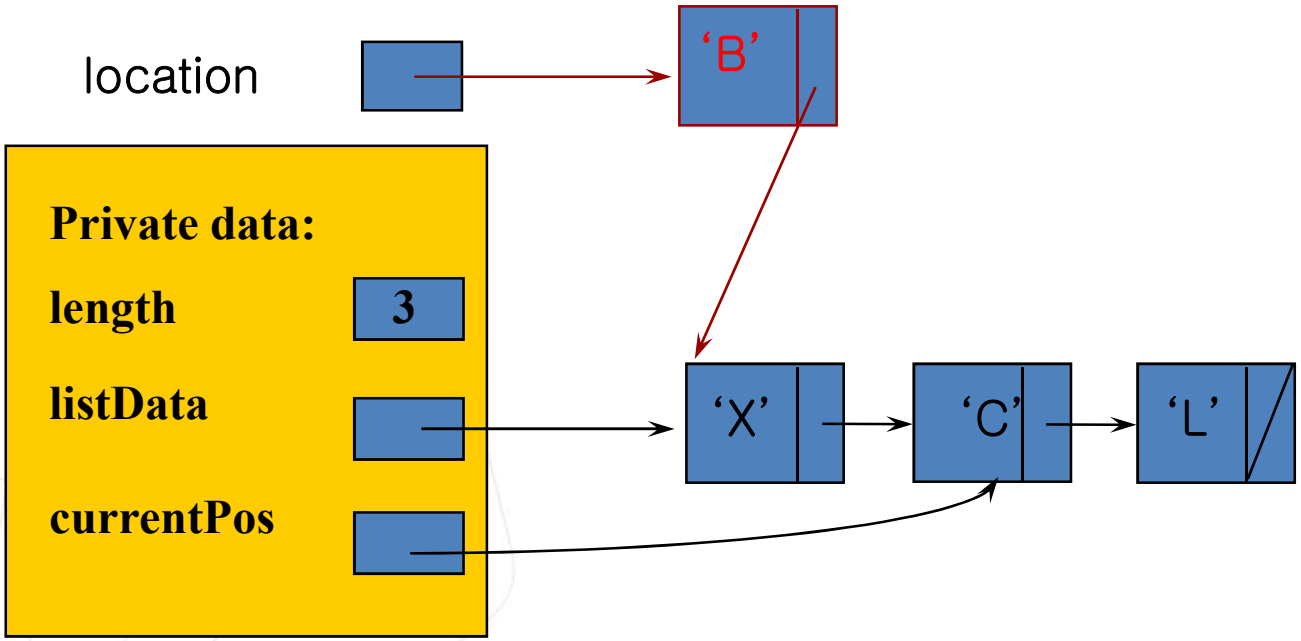


Inserting 'B' into an Unsorted List

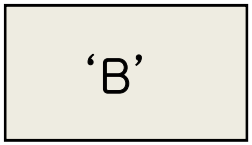


item

```
location->next  =  listData ;
```

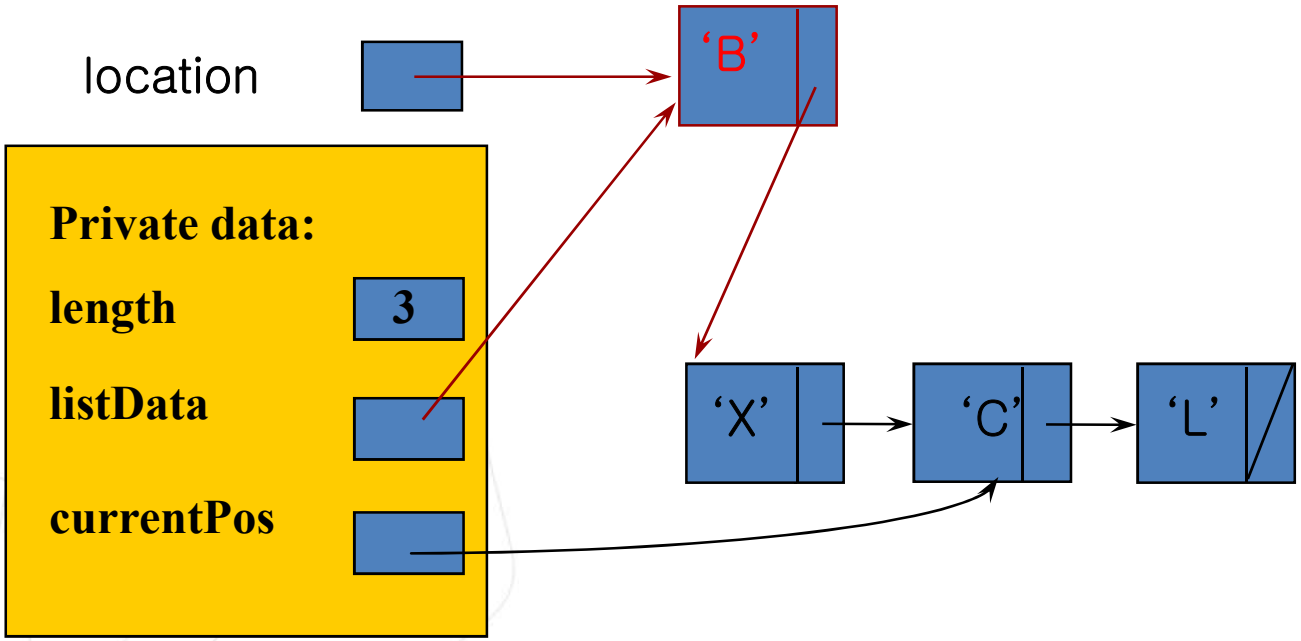


Inserting 'B' into an Unsorted List

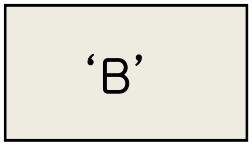


item

```
listData  =  location ;
```

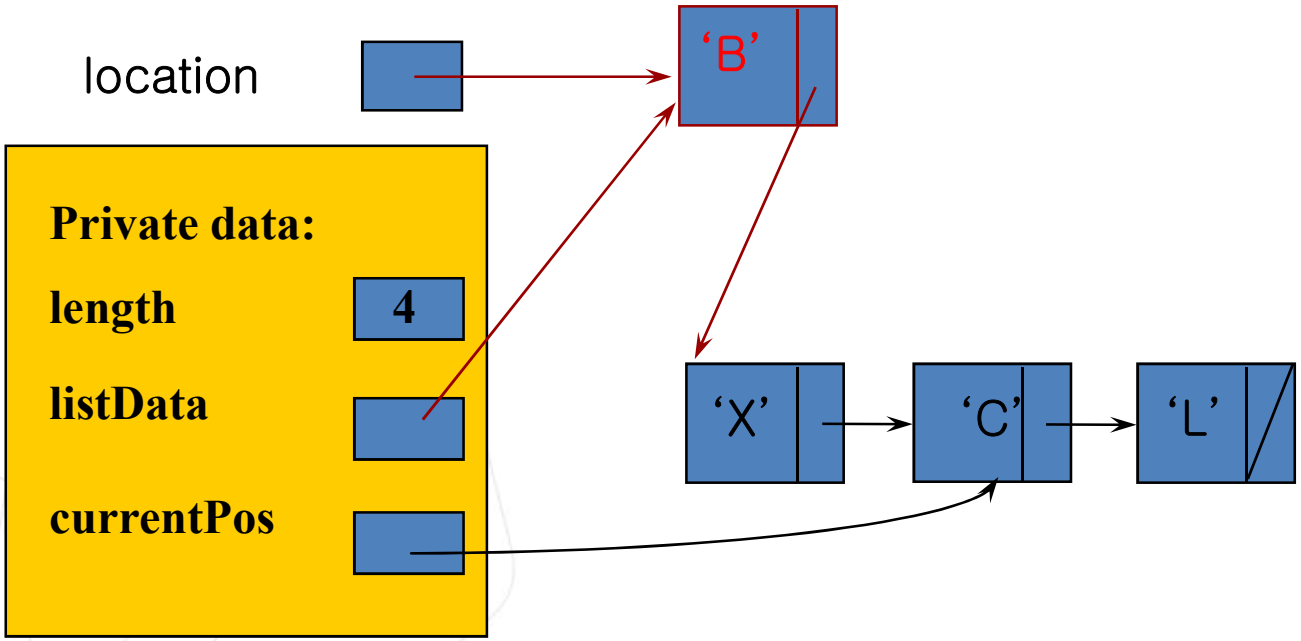


Inserting 'B' into an Unsorted List



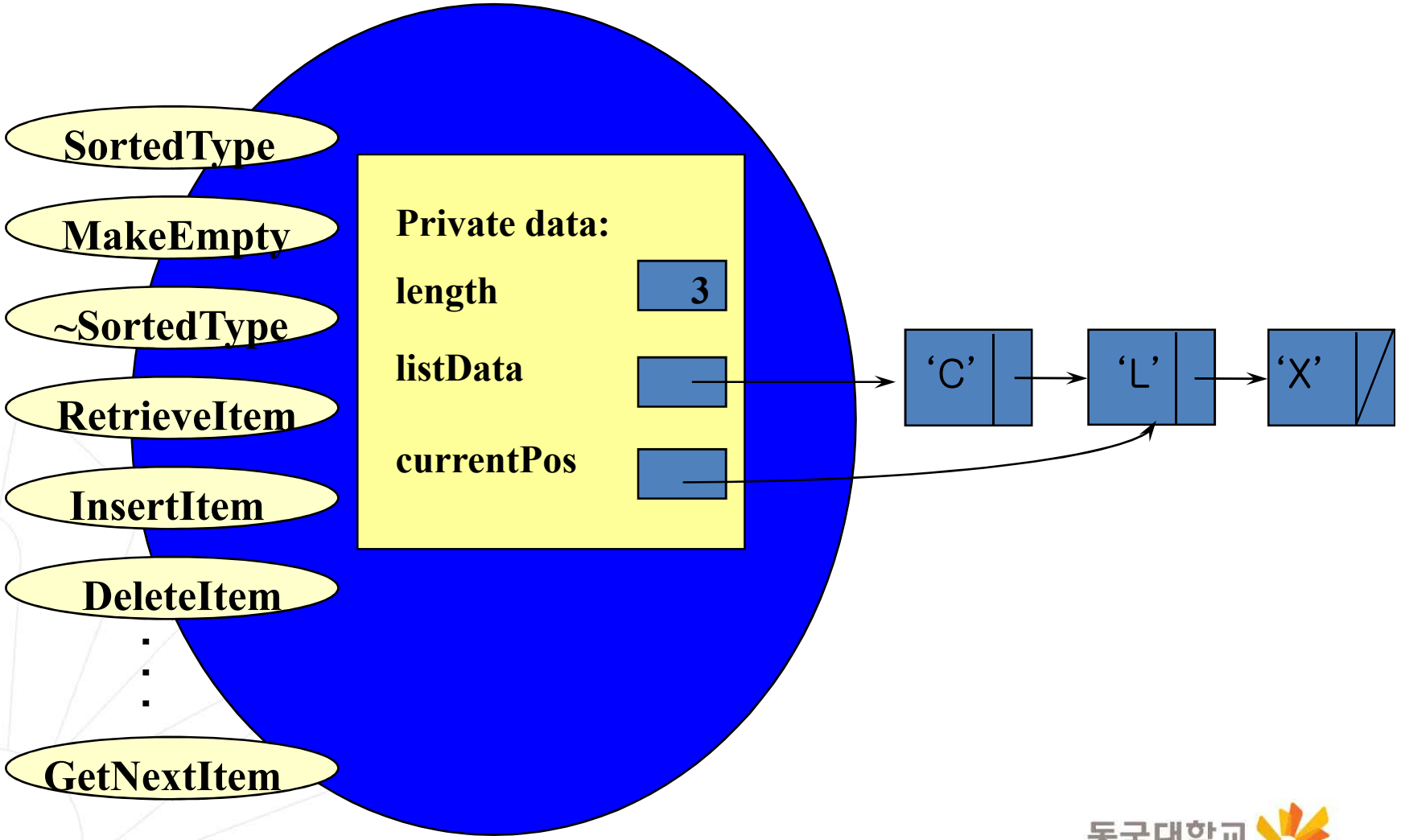
item

```
length++ ;
```



Implementing the Sorted List as a Linked Structure

- class SortedType<char>



InsertItem algorithm for Sorted Linked List

- Find proper position for the new element in the sorted list using two pointers predLoc and location, where predLoc trails behind location.
- Obtain a node for insertion and place item in it.
- Insert the node by adjusting pointers.
- Increment length.

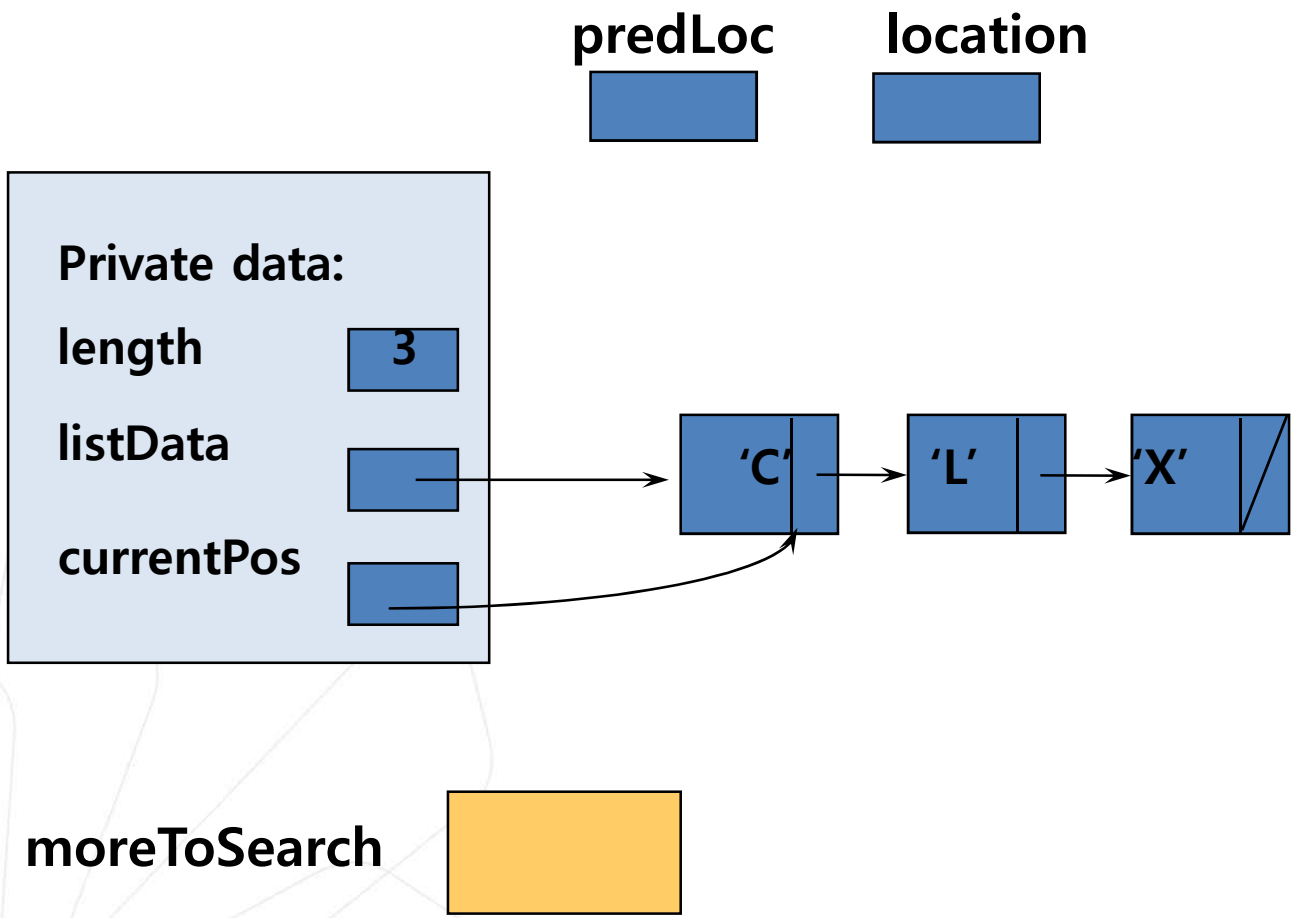
SortedType member function InsertItem

```
// LINKED LIST IMPLEMENTATION                    (sorted.cpp)
#include "ItemType.h"

template <class ItemType>
void  SortedType<ItemType> :: InsertItem ( ItemType  item )
//  Pre: List has been initialized. List is not full.  item is not in list.
//       List is sorted by key member.
// Post:  item is in the list.  List is still sorted.
{
                    .
                    .
                    .

}
```

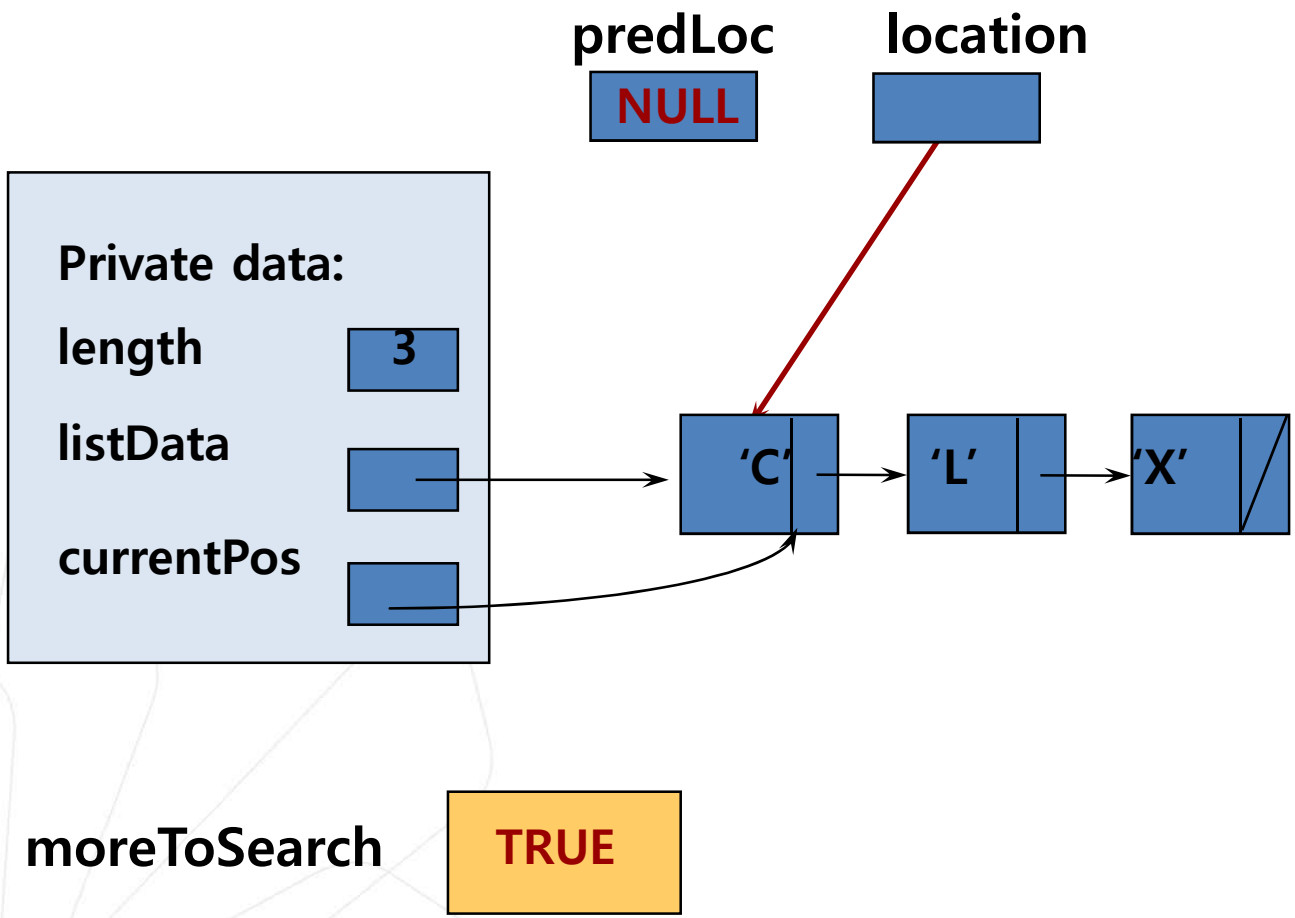
InsertItem()

```
Set location to listData
Set predLoc to NULL
Set moreToSearch to (location != NULL)
WHILE moreToSearch
        SWITCH (item.ComparedTo(location->info))
                case GREATER : Set predLoc to location
                               Set location to location->next
                               Set moreToSearch to (location != NULL)
                case LESS:     Set moreToSearch to false
Set newNode to the address of a newly allocated node
Set newNode->info to item
Set newNode->next to location
Set predLoc->next to newNode
Increament length
```

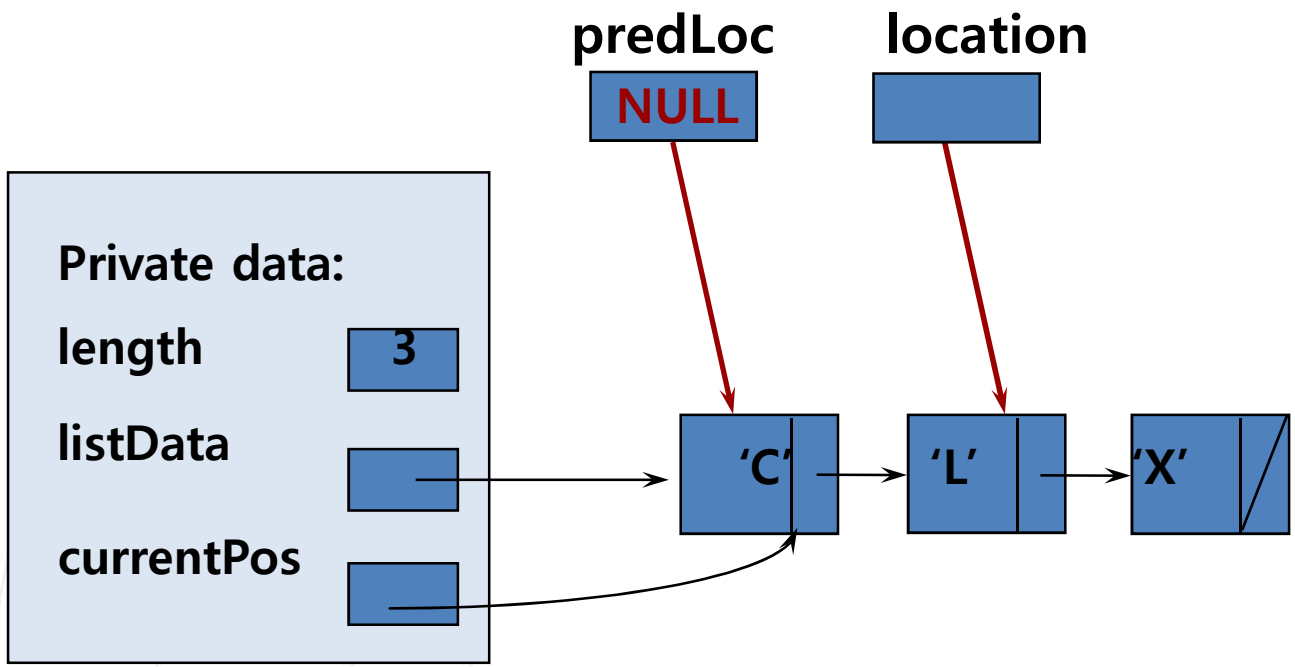
Inserting ‘S’ into a Sorted List



Finding proper position for ‘S’

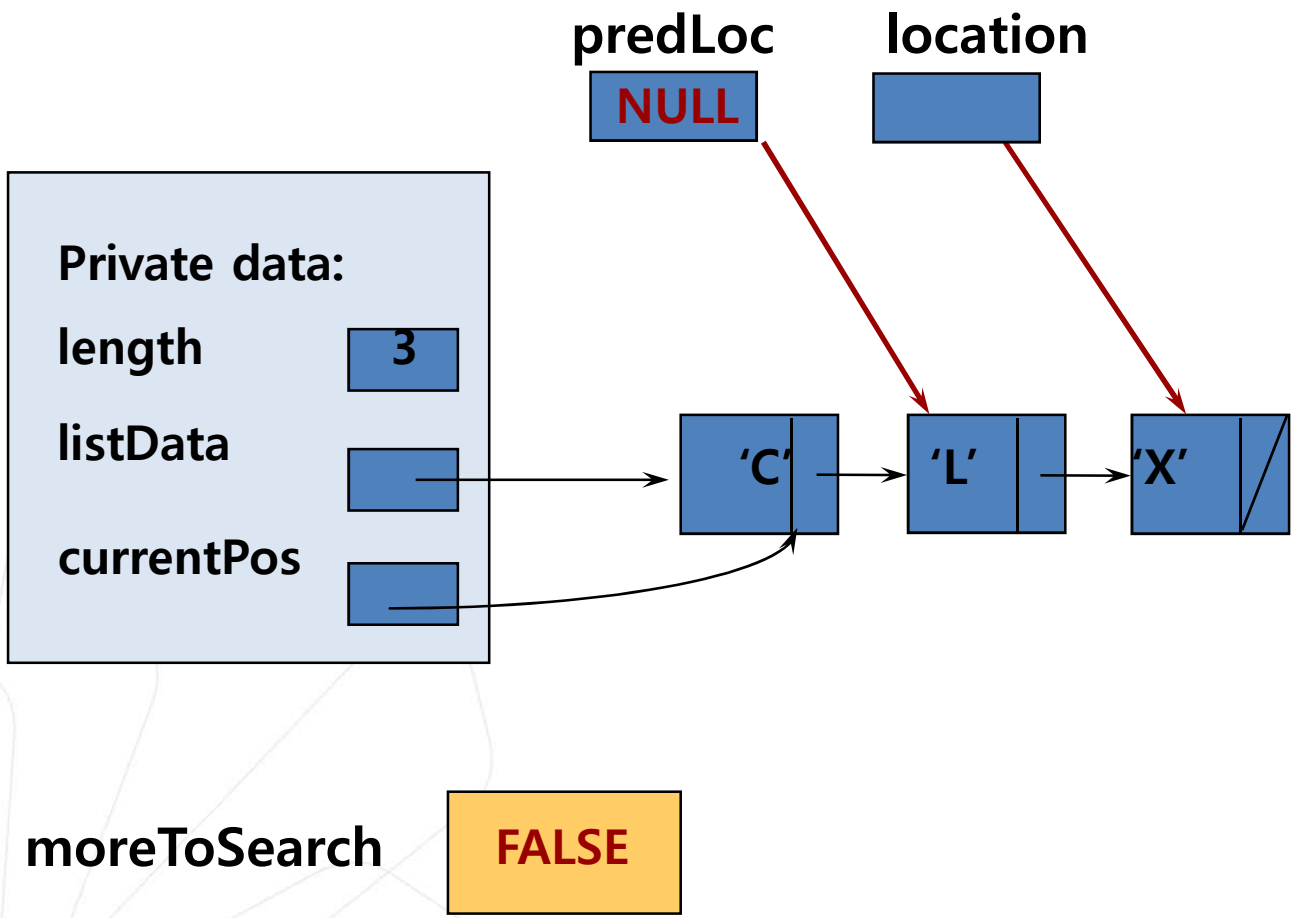


Finding proper position for ‘S’

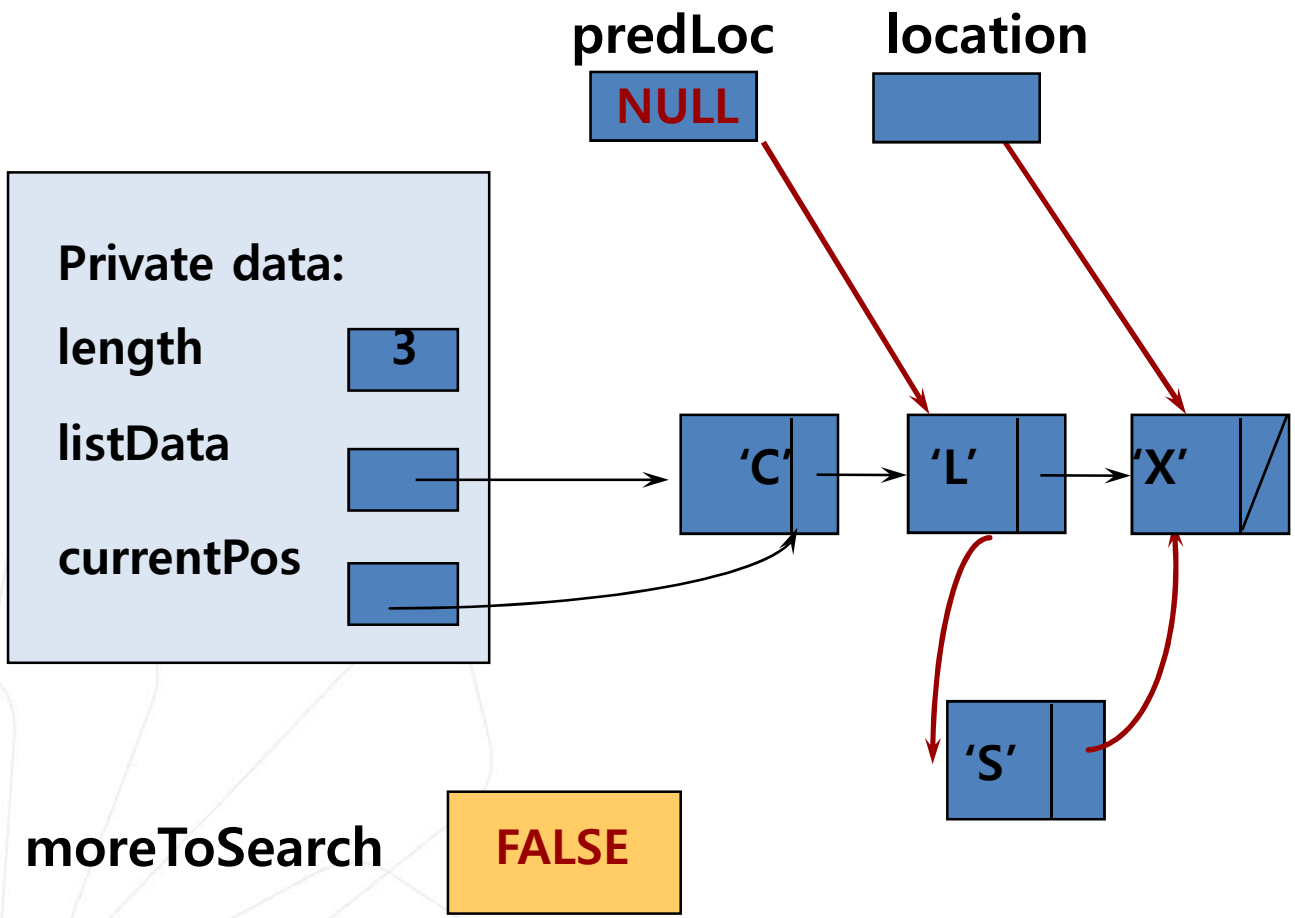


moreToSearch TRUE

Finding proper position for ‘S’



Inserting ‘S’ into proper position



Summary

Array-based or linked representation for

- stacks
- queues
- unsorted lists
- sorted lists

Variations:

- doubly linked lists
- circular lists
- …